Styx Caching via Journal Callbacks

Venkatesh Srinivas
Nathaniel Wesley Filardo

me@acm.jhu.edu, nwf@cs.jhu.edu

Association for Computing Machinery
Johns Hopkins University
Baltimore, MD

ABSTRACT

Styx is a network protocol used in the Plan 9 and Inferno distributed operating systems. This protocol provides a common language for communication within the above-mentioned system. Styx is a simple client-driven, message-oriented protocol. This protocol performs poorly on high-latency links, independent of bandwidth, and has no provisions for caching or server-initiated notifications. Previous attempts at hiding latency have either required replacing Styx or accepting dramatically weaker coherency guarantees.

This paper introduces Journal Callbacks (JC), a mechanism for server-initiated notifications in client-driven protocols, such as Styx. Journal Callbacks allows for these notifications without modification to the underlying protocol.

We implemented a cache for Styx using Journal Callbacks. We attempted to hide latency by caching server responses; notifications are used for invalidation events. Notably, our cache and notification scheme does not alter the Styx protocol; instead, it runs as a side protocol on top of the existing stream. We present data from several benchmarks, showing our cache reduces effective latency comparably to the Plan 9 cfs cache.

1. Introduction

Styx [1] is a very simple resource abstraction protocol. It describes a set of named hierarchical trees of named objects; leaf entries in the tree are named ‘‘files’’; other entries are termed ‘‘directories’’. All objects store a fixed set of metadata. Additionally, files each provide an optionally seekable single stream of bytes. Some Styx servers support creating or removing subsets of their exported objects.

Styx allows more than one outstanding request through a client-controlled "tag". Responses may be uniquely paired to their requesting message since the server will simply copy the tag back. There is a mechanism for request cancellation by tag, but this facility is rarely used.

A Styx connection uses client-specified integers named "fids" to represent live handles to objects. A fid, naming the root of the server’s hierarchy, is derived from an initial attach message. Fids may be walked to traverse the exported file tree, stat-ed or wstat-ed to read or to write metadata, or Opened for subsequent Read and Write requests. Create operations take a fid naming the parent directory as well as the name of the object to be created. Once a fid, Open or not, has finished its purpose, it is Clunked and its identifier is safely available for reuse.

Styx uniquely identifies every version of every object (typically "file") on a server by an entity called a QID. QIDs expose some minimal "type" information, a "path" identifier (essentially object identifier), and a "version" field. Typically, versions are incremented whenever a mutation request (i.e. write, wstat, create, or remove) is successful.

Opened fids *track* the current version of any server-side object. That is, if there are two fids, either from one or two clients, naming a given object, and one fid is used for a Write, then a subsequent Read on either fid will reflect the changes made.

Prior work [2] has demonstrated that operations over Styx can be dominated by latency of the link, which indicates that large performance gains may be had by reducing the number of RPCs that cross the wire. There are a number of ways one might go about this:

1. Redefining the protocol to need fewer RPCs,
2. Intercepting client RPCs and answering from cache before they may go over the wire, or
3. Altering the behavior of clients to eliminate superfluous RPCs.

Previous work falls into the first and second categories. This work is also of the second variety though we believe we are the first to investigate and find opportunities of the third class.

2. Related Work

2.1. cfs(4)

cfs(4) is an on-disk cache intended for use by Plan 9 terminals. It copies data from read messages into an on-disk cache. For subsequent read operations, if the data are already present, cfs responds with cached data. Once per open, cfs will stat an object on the server to check for validity of cached contents. cfs does not cache directory contents nor, by extension, does it attempt to hide any latency for walk operations. Every Walk, Stat, and Read on a directory is simply passed through to the server. Open messages become Stat followed by a open when the cached contents are shown to be stale.

cfs by design eliminates the ‘‘tracking’’ feature of fids described above; that is, opened fids passing through a cfs instance will continue to expose whatever data is in cache, not what is present on the server. It is therefore possible, since cfs does not do readahead or whole-file caching, to see a file’s stream in a state that corresponds to no server version and have to re-open the file to resynchronize.

2.2. Octopus’s Op

Op [2] is a revision to the Styx protocol which batches together operations on the wire to minimize the impact of latency. Ofs, the program which does Styx-to-Op intermediation, may optionally act as a cache. When so doing, it assumes data is unaltered for a brief period of time known as "coherency window" before it will act like cfs and check the remote server for validity.

2.3. Network File System

NFS [3] is a protocol in the UNIX world superficially similar to Styx. At least one modern NFSv3 client provides ‘‘close-to-open’’ cache coherency, similar to that found in cfs(4). When a client has a file open, it is assumed that the client’s cache matches the authoritative copy and that no other client is making changes. When the client closes the file, all still dirty cache contents are written back to the server and the client's kernel issues a GETATTR. If on the next open call, the GETATTR request returns the same value, the cache contents are assumed valid. Concurrent writes, even concurrent appends, are not sensibly supported.

2.4. Andrew File System

The Andrew File System [4] also provides client-side caching. Clients are given time-limited promises of notification should an opened file change. Clients may extend these by actively reregistering their interest with the server. In Coda, an AFS descendent, these callbacks are able to range in granularity from files to entire AFS volumes; see5 . AFS assumes only one concurrent writer and generally writes back to the server only when a file is closed or when the cache overflows; therefore, servers do not call for writeback and there is no inter-client cache coherency.

2.5. Common Internet File System

Microsoft's CIFS [6] supports caching using both "opportunistic" and explicit (byte range) locking strategies. CIFS servers will notify clients of invalidations to open files; we are unclear if this extends to opened, cached files that are not currently open. CIFS allows caches to buffer writes and release them only on server notification of opportunistic lock breaks. Other clients are stalled while the server waits for the owning client to write back. The lock taking operations and notifications are built in to the underlying RPC protocol.

3. Design and Implementation

The next few sections describe the core ideas of JC cfs and then how JC cfs is put together, first the cache controller, then the client-side cache.

3.1. The Design

As mentioned, Styx uses QIDs to uniquely identify every version of every object on the server. The evolution of a Styx file server's state could be described by an append-only log of every QID modified (that is, whose version field changed) or removed. (Such a *journal* need not include creation events; we may assume that the toplevel directory simply exists by offset zero and all subsequent creation events will modify the containing directory.) Each client-side operation can be thought of as having some index into this log; conversely, each offset corresponds to zero or more read operations and exactly one write operaton. If a cache were to read this file and watch for appends, it would know when, subject to network latency, to invalidate cached data and reread from the server.

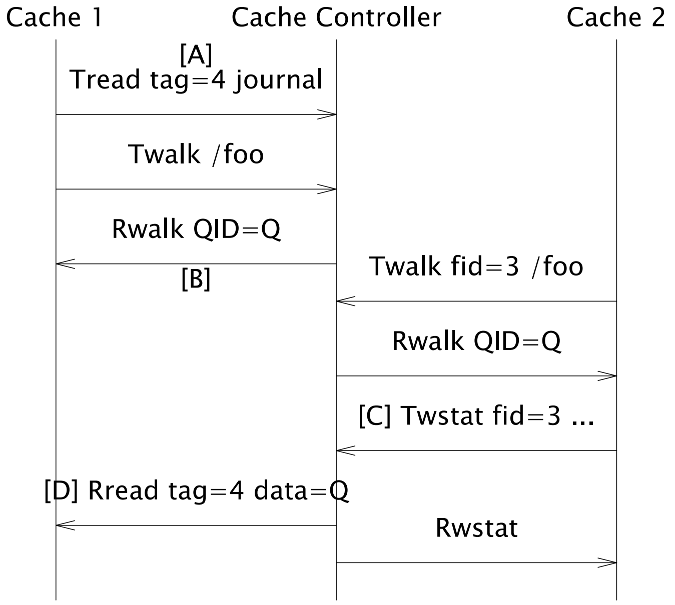
There are four agents in the Journal Callback design: the client, the server, the client-side cache, and the (server-side) cache controller. The client, often the kernel's `devmnt` , and Styx server (e.g. fossil) remain unmodified. The cache and controller act as Styx intermediators: that is, they have two Styx connections and respond to events from each. The cache and cache controller communicate using their own Styx messages over the wire. Additionally, the cache and cache controller are each free to initiate requests of the server. To avoid changing the Styx hierarchy as viewed from the client, we create a parallel hierarchy, using the *aname* feature. This design decision allows us great flexibility going forward.

The cache controller encapsulates all inter-cache information management. It serves to inform one cache when another has successfully carried out a mutation of server state. The cache controller is assumed to sit between all caches and the server. That is, while typically a server is permitted to handle clients directly, in the JC scheme we assume that the server's sole client is the cache controller.

Our cache controller filters the global QID journal to be specific to each connecting cache (which are identified by UUIDs). Every QID reported to the cache is considered cached. Further, it attempts to maintain knowledge of which server data have been seen by the cache even after the cache disconnects; it is possible to indicate to the cache that it has been gone too long and that it must assume that all cached data is out of date.

The cache mediates between a client and the cache controller. It will return cached contents - file, stat, and directory data - quickly when present and believed to be up-to-date. Caches are free to adopt a number of behaviors, including cfs-like behavior or simply blocking client requests, when the journal indicates that they are not synchronized with the server.

An example trace of a cache (1) implicitly registering interest in a file `/foo` and another cache (2) causing the cached contents to become invalid can be seen below. At point A, Cache 1 believes itself to be fully up to date and issues one more read against the journal which blocks. By point B, the cache controller has registered Cache 1's interest - that is, the potential to have cached - the QID Q. The second cache's Twstat operation at point C is forwarded to the server, and if the response is an Rwstat (rather than a Rerror), the cache controller wakes Cache 1 by answering the blocked read (point D) and will forward the success to Cache 2.



For performance and security reasons, one might wish to integrate the cache controller and server. We have not done so largely for ease of implementation and to avoid tethering ourselves to a particular server.

We have not yet implemented a mechanism - such as an append-only write-only file located beside the journals - for caches to notify the controller that a QID has been flushed. Such a mechanism would reduce cache controller memory and unnecessary notifications.

3.1.1. Asynchronous Notifications in Styx

We cannot claim that the core idea, of using a synthetic file to deliver events, is new. From the outset, one of us used the documented behavior of usb(4) audio devices,

> When all values from **audioctl** have been read, a zero-sized buffer is returned (the usual end-of-file indication). A new read will then block until one of the settings changes and then report its new value.

as precedent. We are unaware of any prior implementation of the scheme for caching, however.

3.2. Implementation

As mentioned, both the cache and the controller act as Styx intermediators. Since Styx offers only a single namespace for each of message tags and fids, both of these programs maintain mapping tables so that they can safely rewrite incoming and outgoing requests to avoid collision and, in the case of the controller, can map responses back to the appropriate cache.

The cache and cache controller are implemented in the Limbo programming language for the Inferno operating system; they total approximately 2400 lines of code. The client-side cache is approximately 600 lines, the server approximately 900, and the remaining 800 dedicated to plumbing - mapping structures for QIDs, Fids, and File structures, and infrastructure (module loading, argument parsing).

3.2.1. The Cache Controller

The cache controller is a constructed from a set of concurrent processes synchronizing through message passing. On receiving a connection from a client, the cache controller starts a number of processes to handle the per-client state: *remoteproc, tmsgfd2chan, rmsgfd2chan,* and *sjournalproc.*

Remoteproc exports a path in its namespace as the main filesystem to its client; to do this, it constructs a pipe and spawns an asynchronous exportfs kernel process. It then constructs two processes, *rmsgfd2chan* and *tmsgfd2chan* , to convert reads and writes on the pipe and client file descriptors into Limbo channel messages.

The *remoteproc* accepts Styx T-messages from its client and R-messages from the exportfs kernel process. It dispatches messages to the correct destination, based on a message's fid. Messages destined to the main file system are forwarded along the pipe to the exportfs process; most messages destined to the cache control file system are handled internally and receive a synchronous response. Read requests on journal files, however, are handled by starting a process, *handle_async_cc_read* , which returns data from the journal when it is available.

When a client cache first starts up and connects to a cache controller, it attaches to the cache-control file system and attempts to open its journal file, identified by a UUID. If its journal file does not exist, it attempts to create and then open it. In the cache controller, creating a journal starts another process, sjournalproc, which mediates sending and buffering invalidation messages from the cache controller to the client. Sjournalproc listens on two per-journal Limbo channels; one receives cache invalidation events, the other receives response channels from *handle_async_cc_read* Handle_async_cc_read provides sjournalproc with a channel in response to a client cache read on its journal. If the journal has any outstanding updates, it sends them along the new channel and drops its references to both the events and the return channel. Otherwise, the reply along the return channel is withheld until events are available. In this way, journals can continue to enqueue events even when a client is not attached.

3.2.2. The Client

The client-side cache acts as a Styx server on its standard input/output, for its client, and as a Styx client to the cache controller. The cache is constructed from a set of concurrent processes, similar to the cache controller. These processes, *tmsgfd2chan, rmsgfd2chan, msg2wire, journalproc,* and *scfs,* forward local Styx messages, return responses, and generate journal messages. They also maintain the read and stat cache data structures. *Tmsgfd2chan* and *rmsgfd2chan* are as in the cache controller – they convert Styx messages to Limbo channel message. *Msg2wire* provides synchronization for the connection to the cache controller, so that the main process *scfs* and the journal process *sjournalproc* do not interfere.

Scfs is the cache main process. It receives Styx T-messages from a client, typically Inferno's devmnt, rewrites the FIDs, and forwards most of those messages to the remote cache controller it is connected to. For TStat and TRead messages, it looks up the live file structure by its FID; the per-FID structure points to a per-QID structure, which holds a copy of the file's directory entry and a reference to its read cache. So long as the directory entry and read cache are present, they are used to serve reads entirely locally.

Journalproc is the main journal process; on starting, it attaches to the cache control *aname* and attempts to open its journal via a UUID; if the journal is not present, it creates it. *Journalproc* then enters a state machine, issuing Reads to the journal file and waiting for replies. On receiving a reply, it extracts the encoded QID, looks up the per-QID structure via a hash table, and invalidates both the directory entry and read cache contents for that file, thus keeping the read cache current.

4. Results

As a test workload, as well as a mechanism for ensuring that our code worked, we use our own build process as a benchmark. This involves running `mk` and the `limbo` compiler, reading system headers and our source files, and generating our `dis` executables. The total number and distribution of RPCs for the `mk` workload is provided in the Execution Characteristics section. We hope it provides a typical, read-heavy workload.

Measurements were taken in a few environments:

1. a trans-Pacific link, from a client in Baltimore to a server in Tokyo, Japan. Typical latency in this link was roughly 180 ms.
2. a trans-continental U.S. link, from a client in Baltimore to a server in San Francisco. Typical latency here was roughly 90 ms.
3. with the client and server both on the same machine.

RPC traces were captured with `mount-S` and a tool to capture Styx protocol traces, `statlisten.` For execution workloads, timing data, as reported by `time`, as shown is the average of three runs.

All jccfs measurements were taken in Inferno 4e on a Linux 2.6 host. All measurements of cfs were taken on a native Plan 9 CPU Server.

Results are not directly comparable between the Plan 9 and Inferno systems – the Plan 9 system hardware was different than that of the Inferno systems and the Plan 9 client issues a different number and distribution of RPCs to our cache. RPC counts and percent wall-clock time reduction are perhaps the most useful statistics.

Total RPC counts, for un- and cold-cache behavior

Host	Job	Walk	Clunk	Stat	Read	Write	Open	Create+Rem	TOTAL

Inferno										
(uncached)	mk all	151	83	7	99	49	62	14		465
(uncached)	re-mk	5	3	0	6	0	3	0		17
(uncached)	mk nuke	28	5	0	12	0	5	14		64
(jccfs)	mk all	151	83	7	139 *	48	62	14		504
(jccfs)	re-mk	5	3	0	3	0	3	0		14
(jccfs)	mk nuke	18	5	0	18 **	0	5	14		60
Plan 9										
(uncached)	mk all	151	83	14	97	21	55	14		435
(uncached)	re-mk	5	3	0	4	0	3	0		15
(uncached)	mk nuke	26	5	2	8	0	5	14		60
(cfs)	mk all	151	83	14	43	21	55	14		381
(cfs)	re-mk	5	3	0	4	0	3	0		15
(cfs)	mk nuke	26	5	2	8	0	5	14		60

*: 139 TReads were issued; 96 were asynchronous and to the cache aname; 43 were to the main aname **: 18 TReads were issused; 7 were asynchronous and to the cache aname; 11 were to the main aname

mk all times vs latency

Latency	System	Uncached	Cold	Hot
14 ms	Inferno/jccfs	7.8 s	7.5 s (3.8%)	6.8 s (13%)
90 ms	Inferno/jccfs	49.5 s	43.6 s (12%)	38.5 s (22%)
180 ms	Inferno/jccfs	92.7 s	80.3 s (13%)	73.1 s (21%)
180 ms	Plan 9/cfs(4)	103.1 s	79.4 s (23%)	72.2 s (30%)

re-mk times vs latency

Latency	System	Uncached	Cold	Hot
90 ms	Inferno/jccfs	2.8 s	–	1.8 s (36%)
180 ms	Inferno/jccfs	5.3 s	3.3 s (38%)	2.9 s (45%)
180 ms	Plan9/cfs(4)	3.9 s	2.3 s (41%)	2.3 s (41%)

mk nuke times vs latency

Latency	System	Uncached	Cold	Hot
180 ms	Inferno/jccfs	12.8 s	11.9 s (7.0%)	6.6 s (48%)
180 ms	Plan9/cfs(4)	11.4 s	11.1 s (2.6%)	6.1 s (46%)

We demonstrate a percentage wall clock time reduction roughly in line with cfs(4), though we pay a little bit for our (unoptimized; see future work) journals.

5. Discussion and Conclusions

Our implementation of Journal Callbacks and cache for Styx exhibits similar performance gains in absolute time to the Plan 9 cfs cache. Also looking at the timing measurements in the previous section, as latency rises, both cfs and jccfs scale similarly – their mechanism of operation is very different, however. Cfs reduces the total number of read requests. JC Cfs, while increasing the number of RPCs, serves both Stat and Read requests from its cache.

Compared to cfs or an uncached client, jccfs increases the total number of TRead requests. Why does jccfs exhibit any performance gain over an uncached client, then? The answer is that reads to the cache control aname are not synchronous with RPCs to the main aname. When we described Styx as 'performing poorly' on high-latency links, one the reasons is that Styx clients synchronously wait for RPC responses before sending future messages, making poor use of a network's Bandwidth-Delay Product. The added cost of journal reads and responses make use of this available capacity to enable server-initiated notifications.

An initial implementation of Journal Cachebacks for caching appears to be approximately as effective as the open-to-close coherency cache, cfs. We have shown that our technique is viable and that even a minimal implementation offers real-world performance gains while maintaining the Styx protocol.

6. Future Work

There are a number of avenues of further investigation which merit attention. First, while we have shown that our scheme works, we have not yet shown at least one of its conjectured major strengths. Second, our journal metadata could be exposed and/or enhanced to great effect. Third, we have found, through testing and observation, a few opportunities to improve the behavior of Inferno's Styx client. Fourth, we feel we are in a good position to give well-motivated suggestions for a hypothetical next-generation Styx.

6.1. On-Disk Caching

As a proof of concept, our cache does quite well; however, it is unable to demonstrate a serious advantage of the JC scheme over that of cfs(4): a cache which has been offline may quickly catch up. We expect an on-disk, terminal-side jccfs cache to dramatically reduce the number of RPCs generated at startup relative to cfs(4). Our cache controller already ensures that journals are maintained and populated even after the cache has hung up or closed its journal; the on-disk cache simply would have taken more time than we had.

6.2. Exposing Notifications to Programs

The journal callback mechanism and controller, if present on a given mount, could be made to provide a file or file system monitoring API, similar to Linux's `inotify`[7] Unlike `inotify`, however, a JC-based API would work over networks (NFSv3 and prior do not seem to be sufficiently capable; NFSv4 is) and could be presented as just another kernel virtual server with ctl file, requiring no additional syscalls.

6.3. Exposing More Information to Caches

The current data stream in the journal file contains only QIDs. This is tragically little information, allowing a cache the sole action of invalidation of cached contents, even if it was the reason for the mutation. Every Twrite in fact invalidates the entire file's content as well as the containing directory. This is hardly ideal, so we would like to report "re-QID operations" to caches emitting mutation operations. Similarly, we could report "small" changes to directories (version increment and other stat changes, insertions, deletions) and possibly even files.

It will be easy to accomidate these features by extending our (fortunately not yet standardized and documented) on-the-wire protocol to embed additional data records after each QID in the journal. We note that as long as these data are well framed (by using, e.g., a TLV format), the protocol is extensible without having to update all caches and controllers in lockstep: a cache encountering a metadata field it does not understand may simply revert to invalidating all data corresponding to the QID, as if there were no ancillary fields.

6.4. Changes to `devmnt` or Additional Latency Reductions

We have observed Inferno's `devmnt` to generate unnecessary RPCs, such as

```
; pwd                                  ; cd .
Tmsg.Walk(1,45,26,nil)                 Tmsg.Walk(1,46,45,nil)
Rmsg.Walk(1,array[] of {})             Rmsg.Walk(1,array[] of {})
Tmsg.Open(1,26,0)
Rmsg.Open(1,Qid(16r8,73,16r80),8192)
Tmsg.Clunk(1,26)                       Tmsg.Clunk(1,46)
Rmsg.Clunk(1)                          Rmsg.Clunk(1)
```

For the `pwd` case, a single Tstat request would have sufficed. In the `cd.` case, no messages should have been sent at all. Whether these traces are due to bugs or deliberate simplification of the logic in `devmnt` is unclear. We note, however, that for correctness we can not avoid sending Topen messages over the wire, and so the impact of an intermediator may be lower than the impact of an optimal rewrite of `devmnt`.

We do not currently, but are in a good position to, synthesize Rwalk messages to entries in cache and only instantiate them on demand. Walks that merely clone may be thought of as always in cache. This would eliminate both RPCs in the cd case. For ordinary file objects, we could also merge open requests to keep only one such fid open over the wire.

Further, Tclunk messages may Rerror but the result is not meaningful if so: the fid is no longer valid and the semantics of close() are that it cannot fail when given a real, open fd. Since the kernel knows the openness state of an fd, there is no need for the kernel to wait a full RTT when it emits a Tclunk. We have shown remarkable improvements in performance (as might be expected from the RPC count table, above) by merely generating Rclunk messages in the cache.

6.5. Recommendations for a Next–edition Styx

Based on the implementation effort and measurements done above, we have some small recommendations to make the protocol more amenable to intermediation:

6.5.1. Standardize Identification of Synthetic Files

Currently, Styx does not have a standard mechanism for discriminating between synthetic and real files. By real files, we mean Styx objects which are expected to read back what was written and, in the case of single open or successful exclusive open, report the same contents for each read through start to finish. On the other hand, things like control files, named pipes, and network connections, are "synthetic" – that is, we are not surprised, and do not assume another concurrent user, when Tread at a fixed

Directories do not permit seeking and are generally considered "real". For non–QTDIR objects, we may further require that all reads at the same offset, again with only one active user, return the same contents.

offset returns different data or an error.

This lack of differentiation means that currently care must be taken as to what portions of the namespace (equivalently, what servers) are subject to (caching) intermediation. For example, cfs intermediating an imported /net would almost surely render the /net so imported inoperable.

A previous proposal, to add a bit to the type field of the QID to indicate the syntheticness (or realness) of an object was previously put forward [8] and met with some resistance. There is an undocumented convention that synthetic files have a QID version of zero. We propose that this be a requirement of correct servers. With objects so labeled, our cache and controller will be able to correctly know when to simply pass requests through to the server.

6.5.2. Standardize or Report Mutation Effects on QIDs

Currently, in response to a mutation, our cache controller must walk a fid to the server's file and re-collect the stat information, despite already knowing the mutated object(s) *state.* In particular, this walk is necessary to recover the version field, which is entirely under server control. We therefore suggest that a next-generation Styx standardize on the already traditional behavior of merely incrementing the version on every mutation event. Should this standardized behavior be seen as unacceptable, we suggeset instead that all mutating Styx RPCs be adjusted to return the QID of objects mutated – Rcreate must contain two QIDs, all others just one.

We note in passing that there has been some discussion of making version fields propagate to root. If this behavior were ever adopted, our cache and cache controller could certainly be made to work with it, but since it is not likely to be standardized we propose that servers doing anything atypical with the version field be forced to declare so in their Rversion messages, so that (our) intermediators may either adopt the propagation of updates to root behavior or fall back to more pessimistic but safe behaviors.

7. Acknowledgements

We would like to acknowledge David Eckhardt for getting the idea of journals planted in one of our heads many years ago.

1. Rob Pike and Dennis Ritchie, *The Styx Architecture for Distributed Systems.*
2. Francisco J. Ballesteros, Gorka Guardiola, Enrique Soriano, and Spyros Lalis, ''Op: Styx batching for High Latency Links,'' *IWP9 2007.*
3. *Network File System*, http://nfs.sourceforge.net/.
4. J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, ''Andrew: A distributed personal computing environment.,'' *Commun. ACM* **29**(3), pp. 184–201 (Mar. 1986).
5. L. Mummert and M. Satyanarayanan, *Variable Granularity Cache Coherence.*
6. *Common Internet File System*, http://msdn.microsoft.com/en-us/library/aa365233(VS.85).aspx.
7. Robert Love, *Kernel Korner – Intro to inotify*, http://www.linuxjournal.com/article/8478.
8. Francisco J Ballesteros, *[9fans] QTCTL?*, http://9fans.net/archive/2007/10/539.